The GRUMPS Architecture: Run-time Evolution in a Large Scale Distributed System

The GRUMPS Architecture: Run-time Evolution in a Large Scale Distributed System

Huw Evans, Peter Dickman and Malcolm Atkinson

Department of Computing Science, Glasgow University
17 Lilybank Gardens, Hillhead, Glasgow, Scotland, UK, G12 8RZ

WORK IN PROGRESS PAPER

Abstract

This paper describes the first version of the distributed programming architecture for the Grumps[1] project. The architecture consists of objects that communicate in terms of both asynchronous and synchronous events. A novel three-level extensible naming scheme is discussed that allows Grumps developers to deploy systems that can refer to entities not identified at the time when the Grumps system and application-level code were implemented. Examples detailing how the topology of a Grumps system may be changed at run-time and how new object implementations may be distributed during system execution are given. The separation of policy from mechanism is shown to be a major part of how system evolution is supported and this is made even more flexible when expressed through the use of Java interfaces for crucial core concepts.

1 Introduction

This paper describes the first version of the run-time architecture for the Grumps research project [4]. The Grumps research project is developing techniques and software to automatically collect, manage and analyse large collections of user actions. The project has four interrelated goals: to make it easier to organise experiments that efficiently and as unobtrusively as possible collect and store traces of remote users' actions; to make it easier to analyse the stored data to test ideas about the users' activities and the facilities provided to support them; to discover whether such an approach is effective in improving the quality of distributed information systems; and to trial these issues in specific application areas, such as education and bioinformatics.

The rest of this paper is organised as follows. Section 2 states the main system requirements that have driven the initial design; section 3 describes the major components of a Grumps system and gives its architecture; section 4 describes the three-level naming scheme; section 5 discusses, with examples, how a Grumps system is evolved at run-time; section 6 details some related work; section 7 briefly describes future work and section 8 concludes the paper.

2 System Requirements

The main requirements for the Grumps architecture are: the system should be able to operate across the Internet which requires support for communication through firewalls and via proxies; the topology should be changeable at run-time; and the operations performed by the deployed objects should also be subject to change at run-time.

Support for run-time evolution in Grumps is provided in the two main areas of the naming scheme and in the approach to the design of object deployed into the system. The deployed objects, called GrumpsContainers, are containers for the objects that perform application-level processing over the events that flow around the system. The objects that process these events are called Grumps Units (GUs). Grumps allows these container objects and the GUs to be evolved by sending a GrumpsContainer a control event. The control event contains code to inspect the GrumpsContainer and call its public methods to alter its behaviour and the behaviour of the GU objects it contains.

A Grumps system consists of a graph of network-deployed GrumpsContainer objects. A GrumpsContainer has a location in the network of machines. Into this container are placed the GU objects that perform some part of the application semantics. These contained objects are connected together at run-time by sending connection information to a GU. During system execution, its objects move through a number of well-defined object life-cycle states. For example, the contained objects are created, deployed into a container, connected to other objects, have their implementations changed during computation, are removed from the deployed system and are then finally archived or destroyed. In order to be able to support such a system requires a naming scheme that can refer to such entities (in terms of their location and functionality) and a GU architecture that supports evolution as one of the most common operations performed in the system. The common operation is to evolve the objects that a container holds, although the architecture also supports the ability to add and remove container objects and also to evolve a container's implementation.

The Grumps architecture builds on the ideas contained in the DRASTIC project [2, 3].

3 Major Components

3.1 Input and Output Channels

In Grumps, events are communicated via input and output channels. An input channel is defined to be the receiving end of a communication line, which can be read to receive events. An output channel is the sending side of the communication

[1] Grumps stands for Generic Remote Usage Monitoring Production System.

line. Events that are written at the sending side are read at the receiving side and one output channel is connected to exactly one input channel. The sending side and the receiving side can be in the same process or in different processes, on different machines. The phrase "input channel" refers to the receiving end (server-side) of a communication line and "output channel" refers to the sending side (client-side) of a communication line. The phrase "event channel" refers to a whole channel, and does not distinguish between either end.

3.2 GrumpsContainers, Grumps Units and Grumps Events

The two major components in the Grumps architecture are the GrumpsContainers and the Grumps Unit (GU) (figure 1). A GrumpsContainer is an object that encapsulates a number of GU objects. Each GU object is referred to by a number of input channels and the processing object may refer to a number of output channels. Each GrumpsContainer object can be communicated with via its single control event channel. Input and output channels communicate in terms of Grumps Events (GEs) which are sent asynchronously. The control channel is used to change the container at run-time. Control events are sent to the control channel synchronously[2].

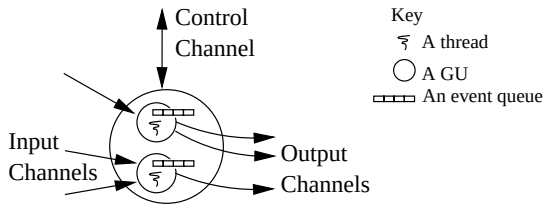

Figure 1: An Implementation View of a Grumps Unit

A Grumps event carries with it information on when the event occurred, which object originally sent it and some event specific information. Each event arriving on an input channel is placed into a FIFO queue, one queue per channel. This queue is managed by an object that is responsible for reading the objects from the queue, processing them in some way and (possibly) sending them out on an output channel. GUs can be combined into graphs of GUs and it is the intention of the Grumps project to be able to treat a particular graph of GUs as a single GU. This will allow users of the system to reuse components, building sophisticated experiments from a collection of GUs with well-known behaviour.

A number of specialised GUs have been identified. Some of these are: autonomous_logic, which takes events and emits control events; operational_store, which makes events persistent; and compress, which takes a single event or a stream of events and compresses it or them into another form. By combining these GUs into graphs it is hoped that an experimenter can quickly and easily build experiments.

3.3 Experiments and Environments

One intention of the Grumps architecture is to be able to make it easy to organise experiments. A Grumps experiment consists of a number of computers that are connected via a network. Each machine runs a number of environments and within an environment there are a number of GUs in their containers. A Grumps experiment and its environments are named entities in Grumps, e.g., the environments are named operating system processes. An example of this part of the architecture is given in figure 2.

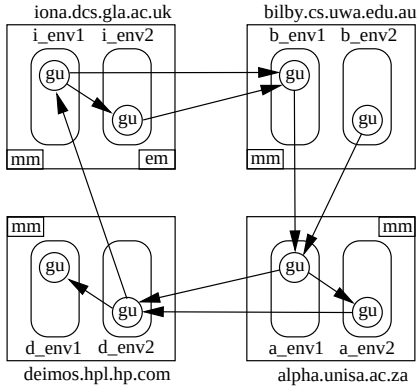

Figure 2: An Example Grumps Distributed Architecture

Here we have four machines: `iona.dcs.gla.ac.uk` is in Scotland; `bilby.cs.uwa.edu.au` is in Australia; `alpha.unisa.ac.za` is in South Africa and `deimos.hpl.hp.com` in on the west coast of America. Each machine is hosting two environments, `i_env1` and `i_env2` in the case of `iona.dcs.gla.ac.uk`. To keep the diagram simple, each environment has a single GU (the control channel is not shown) which has a number of input and output channels.

`iona.dcs.gla.ac.uk` supports another kind of GU, called `em` on the diagram, which is the `ExperimentMonitor`. This GU is responsible for providing facilities that are common to the whole experiment. This includes serving Java class files and providing a name service for the objects involved in the experiment[3]. When the `ExperimentMonitor` is executed it is given an experiment name, which is a simple string allocated by the administrator of the system. Each environment that is started with that experiment name is considered to be a part of that experiment.

On each machine a GU, called `mm` on the diagram, is running. This is the `MachineMonitor` and it is responsible for controlling the environments on that machine, e.g., such as starting a new one. `mm` is running inside an environment called `MachineMonitor` which has been started within the context of an experiment. The `MachineMonitor` and `ExperimentMonitor` GUs are held in their own GrumpsContainer objects and they are executing in separate environments.

4 Naming in Grumps

In setting up a graph as described in figure 2 the software needs to be able to refer to the entities in the system. For example, the GU that is running in `i_env1` must be able to refer to the GU that is running in `b_env1` in order to establish a remote reference to it. This is accomplished with the novel Grumps physical naming scheme.

In Grumps, the experimenter needs to be able to name many different kinds of entity, for example, GrumpsContainers, their GUs, environments, machines, files, data collection

[2]Communication is synchronous to the control channel as it is generally useful to be able to get a result object back as a result of sending in a control event.

[3]Given that this experiment involves computation on four continents the `ExperimentMonitor` facilities will have to be federated at each site. This is an area for future work.

devices and so on. An extensible physical naming scheme is defined that allows the experimenter to name such entities and be able to create new naming schemes to refer to entities that are yet to be identified.

4.1 Extensible Physical Naming Scheme

The general form of the Grumps extensible physical naming scheme consists of four parts: a naming scheme identifier; a naming scheme interpreter; naming scheme information; and freetext to allow for expansion within the naming scheme.

The naming scheme identifier tells the naming-system what kind of name this is, e.g., whether it refers to a GU or a file. The naming scheme interpreter is information on which language-level code should be used at run-time to interpret this naming scheme, e.g., how to get hold of the machine name if the naming scheme is for a GU name. Next is the information peculiar to the naming scheme which consists of information to uniquely identify the entity in question.The freetext is an open-ended string that can be used to add extra information to the name that is not directly catered for in the naming scheme with a given interpreter.

The naming scheme is a physical one because information is built into the names that describes the physical location of the entities that they refer to. For example, for a GU's control channel, the physical information is the name of the machine and the port number on which it is listening.

4.1.1 Referring to an Event Channel

The particular physical naming scheme described here as an example is used by the run-time system to refer to the event channels that are contained within a GU. An event channel (EC) has an output side and an input side. These two sides may be in the same environment, they may be in different environments on the same machine or they may be in different environments on different machines. In addition, the input side may be within the same administrative boundary, or it may be in a completely different domain, behind a firewall or only reachable after contacting a proxy first.

To be able to refer uniquely to an event channel at run-time, the Grumps communication system needs to know two pieces of information: the name of the machine and the integer value of the port on which the EC is listening for incoming events. Three other pieces of information are added to this; the experiment and environment names[4] are added for human users, and a protocol identifier is added to distinguish the kind of inter-GU communication being performed. These five pieces of information constitute the naming scheme information for an event channel which is written thus:

```
/FW/MyExperiment/(130.209.240.35:20000):NameServer/
```

Figure 3: EC Naming Scheme Information

This name describes the receiving end-point of an event channel (which may or may not be active) on the machine whose IP address is `130.209.240.35` (`iona`), listening at port 20000 in the environment called `NameServer`, which is part of the `MyExperiment` experiment. The `FW` tag says that in order to talk to this GU, the communication from the sender needs to pass through firewalls. The above name is prefixed with a Grumps extensible name as shown in figure 4.

```
grumps::ec.name.physical:(java:ECPhysicalNameParser)/
```

Figure 4: EC Extensible Physical Naming Scheme Prefix

`grumps::ec.name.physical` is the naming scheme identifier for the physical event channel. `grumps::` indicates that this is a Grumps name and `ec.name.physical` says that the rest of this string is a physical event channel name. The string `(java:ECPhysicalNameParser)` indicates that the whole physical name was generated by an instance of the Java class `ECPhysicalNameParser` and that to interpret the name an instance of that class should be used.

The extensible physical name scheme allows Grumps programmers to provide naming schemes that are tailored to the specifics of the entity being referred to and to refer to object-kinds that are yet to be identified.

4.2 Logical Names

The above physical names have one serious drawback. If the GU with the event channel that is running on `iona` is stopped and, after a time, restarted, it may not be able to be use port 20000. Another EC may have been started while our GU was down and this other EC could have been allocated port 20000. If this is the case, our GU could run its EC on another port. However, the name given in figure 3 is now useless as it refers to the wrong event channel. Grumps defines a logical naming scheme to abstract over the information at the physical level.

A logical name is a pair consisting of the experiment name and a 1024 bit value. Associated with a logical name is at most one physical name or no physical name. A logical name can possibly have no physical name associated with it as the logical name has been allocated but the exact details of the objects physical location are yet to be defined. The `ExperimentMonitor (em)` in figure 2 contains a name server which can allocate a logical name given a physical name. Once allocated, the logical name is considered to exist forever, i.e., it cannot be reused. The logical name is always associated with the service provided by the entity referred to with the physical name. This service would be one of the fundamental kinds of GU identified in section 3.2 or it may be one of the `MachineMonitors` or it may be a file or a machine. When a programmer has a reference to a particular logical name they need to be confident it refers to the service they think it does, and not some other (potentially completely different) service. For this reason, the logical name cannot be reused. Therefore, it is safe for it to become part of a GU's persistent state, should that be required. This use of logical names in Grumps is akin to the use of `ServiceIds` in Jini [1].

When a logical name contains an event channel physical name (for which there is an input channel), the logical name can be considered to be the output channel. In practice the logical name is presented to the communication layer, the physical name is extracted and a socket[5] connection is opened to the input channel described in the physical name.

[4] Future versions of Grumps will consider whether a GU and its contained objects can participate in more than one experiment at a time. If this is supported, the design of this naming scheme will have to be revised.

[5] Grumps is using sockets for event channel communication as they are independent of any other transport protocol, e.g., RMI or Jini, and they provide a convenient building block onto which future communication technology can be built, such as that required to traverse firewalls and proxies.

4.2.1 Using Logical Names

There are three main operations performed using logical names: associating a physical name with a new logical name; using a logical name to gain access to the physical entity; and reassociating an existing logical name with a new physical name. To discuss these cases within the context of a physical event channel name, assume that the event channel is referred to with the EC naming scheme given in figure 3 and that it is associated with the following logical name object:

(MyExperiment:1330515575...)/(130.209.240.35:20000)/

Figure 5: Using a Logical Name with a Physical Name

where `(MyExperiment:1330515575...)` is the logical name component and `(130.209.240.35:20000)` is the (abbreviated) physical component.

To allocate a new logical name, the following occurs. When the GU on `iona` was started, its EC was allocated the port number 20000 by Java. The `em` name service was then contacted to allocate a logical name and associate the physical name with it. A copy of the logical name/physical name association was kept at the name server and a copy was passed back to the object that represents the event channel that is running on `iona` which is listening on port 20000 for incoming events.

This kind of logical name is then used to communicate with an event channel. Assume the logical name has been handed out to another GU, which is running somewhere else in the network. To open a connection to the EC on `iona`, the logical name is presented to the Grumps communication facilities. The physical name is extracted and is used to establish a socket connection to `iona` on port `20000`. The logical name is passed down the socket, together with the event being transmitted. In the EC object on `iona` the logical name it has been passed is compared with its own. If they are equal[6] the communication can proceed, otherwise the socket is closed and no more communication takes place. This check ensures the client knows it is talking to the EC for which the logical name was allocated, and not a different channel which just happens to be listening on the same port. It is possible that the code listening on the port is not GU code, but could be something else entirely. If this is the case, it is assumed that the code will not understand the serialized objects it is sent and will close the connection. It is also assumed that the code does not perform any action that could adversely affect the Grumps application.

We now assume that the GU and its EC are stopped and restarted on a different machine, in a different environment, and the EC is listening on a different port. When the GU is started in the new environment, the EC object must first of all contact the `em` name server and reassociate its new physical name with the logical name stored in the name server. However, the GU running on the other machine has a copy of the logical name with the old EC physical name in it. When it tries to open a communication socket to the old address, it will either be explicitly rejected as described above or it will fail as no EC is listening on that socket. This client-side error is presented to the programmer as an exception and the name server is contacted, passing in the logical name. The experiment name and the 1024 bit quantity inside the submitted logical name are used to lookup the previously registered version of the logical name, which now contains the new physical name. The result of this lookup is passed back to the client. This logical name is then presented to the communication layer and a successful connection, via the new physical EC name, is made to the remote EC running on the new machine.

[6] Two logical names are equal if the experiment name Java strings are equal and the two 1024 quantities are equal.

4.3 Finding Logical Names

Given the description of the naming mechanism above, the only way to gain access to a logical name was to have been passed a copy of one from another piece of code. As the logical names are logical, they are meaningless to a programmer and so are not suitable to be used as keys to the name service's logical name lookup service. It does not make sense to allow logical names to be looked up on the basis of their currently defined physical name as the logical name is meant as an abstraction over the details of the physical location of an entity. Therefore, the Grumps system defines another way of looking up logical names based on `Attribute` objects.

An `Attribute`[7] object is an object that provides human interpretable information to be associated with a particular logical name. An `Attribute` object may be something like a simple name, such as `Configuration Information` or `Data Collection Device`. When a logical name with its associated physical name has been allocated by the name server, it can have the logical name exposed via a set of `Attribute` objects. For example, a logical name could be exposed via the single named attribute object called `Data Collection Device`. Some other piece of code on the network can then perform a lookup in the name service, passing in, via an array, an attribute named `Data Collection Device`. The result of such a lookup is an array of logical name objects that match this array of attributes.

5 Evolution

This section describes how a GU and the objects it contains may be evolved at run-time. This is explained by considering how a new input channel may be created inside a GU (by sending in a new event processing object) and then how this event processor may be replaced with a new one. During system execution, the GU may be receiving control events while also handling a number of concurrently executing event processing objects. During this activity, code in another environment can send into the GU a control event. This control event can affect the operation or state of the GU itself, or by calling methods on the GU's interface, can gain access to one of the event processing objects and alter its implementation or data.

5.1 The GU Control Interface

This section describes the control interface to a GU in more detail to show how the topology of a GU graph and the semantics of a GU can be updated at run-time. A GU has a single control event channel and makes available a single public method for incoming control events:

[7] The approach to using `Attribute` objects in Grumps borrows a lot from both the ideas of using attributes in Jini [1] and the general idea of a trader as defined by ANSA [5] and is therefore not described in great detail.

```
void receiveCtrl(ControlEvent e) throws GrumpsException;
```

Figure 6: GU method for Receiving Control Events

`ControlEvent` is an interface and instances of classes that implement this interface are passed to the GU along its control channel[8]. A `ControlEvent` object defines two public methods:

```
void apply(GU gu);
void setSocket(Socket socket);
```

Figure 7: ControlEvent Interface

The `setSocket` method is used by the receiving-side of the communication facilities, so that the received `ControlEvent` object can send information back to the client-side that originally sent the `ControlEvent`.

The `apply` method is the cornerstone for supporting the ability to change a GU at run-time. When the GU receives a `ControlEvent` instance (`e` in figure 8), it is passed to the GU's `receiveCtrl` method and the GU executes the `apply` method[9]

```
e.apply(this);
```

Figure 8: Applying the ControlEvent to the GU

Thus, the reference to the GrumpsContainer is passed to the `apply` method that is defined on the `ControlEvent` object. Whatever code has been provided in the implementation of the `apply` method is then run, and together with its reference to the container object, the public methods of the container may be invoked. The GrumpsContainer defines a number of methods that allow for control of itself directly or give access to the GU objects that it contains. For example, when replacing one GU with another, the new GU may require access to the old GU specific state and the new GU may want to ensure all the queued events are flushed out. The GU interface is required to expose its own specific state and it provides an API to gain access to the individual event processing objects, which in turn provide an API to flush their queues[10].

In this way we have decoupled the policy for affecting a GU at run-time from the mechanism used to accomplish it. It is impossible to foresee all the operations that could be required to be performed over a GU. Therefore, it does not make sense to put any policy defining operations into the code for a GU. Once the GU had been deployed, it would be very difficult to change that policy. With this mechanism, the policy can be defined in a class that implements the `ControlEvent` interface and we can transfer this policy object to the GU in question. The GU then executes the code in the `apply` method which affects the GU's operation via its public methods.

[8]The control channel has been registered in the `em` name service using a set of attribute objects that uniquely identify it. This is currently implemented using an instance of the class `ControlChannelAttribute` that indicates the kind of channel this is and several named attributes that describe the experiment, environment and machine name and port number on which the control channel is listening.

[9]Although security is not a primary goal in Grumps, some checks can be performed by the GU on reception of a control event. For example, `receiveCtrl` could reject all control events that were not instances of a particular type. Considering the security issue in terms of cryptographically secure control events or some other mechanism may be an area for future work.

[10]Further developing the GU interface and that of the event processing objects is an area of future work.

5.2 Providing a New Input Channel

A GU in Grumps can support a number of input channels. To create a single new input channel in a given GU a `ConnectionRequestControlEvent` is sent to the GU to which the new input channel should point. This kind of control event defines the `apply` method given in listing 1 (constructor not shown that assigns to `po` and `exposer`).

```
public class ConnectionRequestControlEvent extends ControlEventImpl {
  ThreadedEventProcessingObject po;
  Exposable exposer;

  public void apply(GU gu) {
    if (!(gu instanceof GUContainer))
      return;

    GUContainer grumps_unit = (GUContainer) gu;
    GUConnection connection = new GUConnection(0, po, exposer);
    grumps_unit.addConnection(connection);
    po.start();
    connection.start();

    writeObject(connection.getLogicalName());
}
```

Listing 1: Providing a New Input Channel

`GU` is the interface all GU implementations must implement, `GUContainer` is a top level class that implements the `GU` interface. `GUContainer` provides the GrumpsContainer functionality. We first of all check that the object passed to `apply` is the kind of object we are after. This is so the assignment to the `grumps_unit` object is successful. The next line then creates an object called `connection`. This is the object that manages the new input channel within this GU. The `po` object defines how incoming events on this channel are processed. The `exposer` object defines how this new event channel will expose itself to the `em` name server, using a set of attributes, logical name and physical name as described in section 4. The value `0` means the Java system should allocate the port number on which this connection will listen. The new connection is then added to the GU in the `addConnection` call. The next two lines cause the `po` and `connection` objects to start their threads (see below). The last line returns to the client code (that sent the control event) the logical name of the connection object that was just created. This logical name is then used at the client end to communicate with this connection. Once the logical name is returned to the client side, the new connection has been added and events can be sent to it. These events will arrive in the `po` object which processes them and (possibly) sends them on an output channel, or to some other destination, such as a disk.

5.2.1 Thread Programming

When a new connection is created, two threads are started. The `connection` thread in listing 1 is listening to the port for incoming connections. When one is received the event is read from the socket and passed to the `po` object. The `po` object places the event into a queue and calls its own `notify()` method. This causes the thread portion of the `po` object to wake up from a previous call to its `wait()` method. The thread then takes the next event out of the queue, processes it according to the code defined in the `po` object[11] and (pos-

[11]Processing the event may generate other events, which can be sent to output channels.

sibly) sends it out on an output channel. Access to an output channel is achieved in the `po` object by gaining a (number of) logical name objects and opening communications with them, sending the processed event, or possibly a new event to the receiving EC.

As the code for interacting with the GU is held in the control event object, the policy for how to create the thread objects and start them can be changed over time, by providing new classes that implement the `ControlEvent` interface.

Thread classes in Grumps implement the interface `GrumpsThread`. This provides additional control over the thread, specifically it provides a method `void exit()` that instructs the thread that it must shut down. The `po` object above overrides this method to provide management of its queue of events. For example, when `exit()` is called on `po` it may want to drain its event queue. The code to do this can be provided in the `exit()` method.

5.3 Updating an Input Channel

The `GUContainer` implementation of the `GU` interface also defines a method `getConnection()` which takes a single logical name as an argument. This method returns an object of type `GUConnection`, an example of which is the `connection` object in listing 1 above. This is how the semantics of a GU and its processing of events on a particular event channel is updated at run-time. This is achieved by sending an instance of the `UpdateConnectionControlEvent` class to the control channel. The `apply` method on this control event does the following:

```
public class UpdateConnectionControlEvent extends ControlEventImpl {
  LogicalName logical_name;
  ThreadedEventProcessingObject po;

  public void apply(GU gu) {
    if (!( gu instanceof GUContainer))
      return;

    GUContainer grumps_unit = (GUContainer) gu;
    GUConnection connection =
            grumps_unit.getConnection( logical_name );
    connection. setProcessingObject (po);
    po. start ();

    writeObject (null );
  }
}
```

Listing 2: Updating an Input Channel

This time a reference to the `GUConnection` object contained inside the `grumps_unit` is retrieved using the EC `logical_name` object which is part of the state of this control event. The `GUConnection` object then has its `setProcessingObject` method called, passing a new `po` object in as a parameter. The old `po` object that is inside the `connection` object has its `exit()` method called and the new `po` is then assigned and its `start()` method called. In this case no information needs to be sent back to the client side as the `connection` object has not been changed, thus its logical name remains the same.

6 Related Work

There have been many systems that allow the topology of a distributed computation to change at run-time, such as the ongoing work on Darwin [6] at Imperial College, London. However, the Grumps architecture allows the semantics of the system to change, without having to change its topology. The closest related work in terms of architecture is Jini [1]. However, the Jini work supports highly dynamic networks of services. The Grumps work is focussed on building graphs of interconnected components and evolving them at system runtime in an environment where communication through firewalls and via proxies has to be supported.

7 Future Work

The Grumps project is a three year project and the first six months have elapsed. The basic communication and GU evolution infrastructure is complete, including the three-level naming scheme, the name server and the event mechanism used for communication and code replacement. The object lifecycles of the GU, event processing objects and other classes, such as `LogicalName` and `GrumpsThread` need to be considered in terms of the definition of their interfaces, their default implementations and documentation. Other members of the Grumps team will be providing tools to allow an experimenter to define the topology of their experiment, and then deploy, execute, and tear it down again.

8 Conclusions

This paper has described the first version of the run-time distributed programming architecture for the Grumps project. The architecture and its main components have been described as well as its three-level extensible naming scheme. How the system can be evolved, both in terms of its topology and in terms of its semantics have been described. The separation of policy from mechanism is of crucial importance when providing an environment in which evolution can be supported and this is made even more powerful by the provision of a core set of interfaces that capture the main mechanism. A default policy for this core set of interfaces is provided by a set of classes that implement those interfaces. `ControlEvent` objects contain the policy for updating a GU at run-time.

For more information on the Grumps project, see the Grumps website `http://grumps.dcs.gla.ac.uk`.

9 Acknowledgements

The authors would like to thank the members of the Grumps team for providing the necessary environment in which to perform this work and the UK EPSRC (GR/N38114) for providing the funding.

References

[1] W. K. Edwards. *Core Jini*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1999.

[2] H. Evans and P. Dickman. DRASTIC: A run-time architecture for evolving, distributed, persistent systems. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 243–275, Jyväskylä, Finland, June 1997. Springer.

[3] H. Evans and P. Dickman. Zones, Contracts and Absorbing Change: An Approach to Software Evolution. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '99)*, volume 34 of *SIGPLAN Notices*, pages 415–434, Denver, Colorado, USA, Oct. 1999. ACM.

[4] The Grumps Project. http://grumps.dcs.gla.ac.uk/.

[5] A. Herbert. An ANSA overview. *IEEE Network*, 8(1), Jan./Feb. 1994.

[6] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989, 1995.